PRIFYSGOL ABERYSTWYTH UNIVERSITY

CS10720 Problems and Solutions

Thomas Jansen

Today: Multi-Dimensional Dynamic Arrays Recursion and Merge Sort

March 3rd

Plans for Today

One-Dimensional Dynamic Arrays

Problem Create something like `double a[n];`
where the value of `int n` is determined at run time

Solution `void *malloc(size_t size)`
for allocating the space you need and
`void free(void *ptr)`
for releasing allocated space (i. e., giving it back to OS)

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */
int main(void) {
  double *a;
  int    n;
  printf("Enter number between 1 and 100:  ");
  assert( scanf("%d", &n) == 1 ); /* reads 1 number; stops program if this fails */
  assert( (n>0) && (n<101) ); /* stops program if n not between 1 and 100 */
  a = (double *)malloc(sizeof(double)*n); /* allocates space for a */
  assert( a!=NULL ); /* stops program if space was not available */
  a[n-1] = 42.12; /* stupid example */
  free(a); /* frees space used for a again */
  return 0; /* exits program without error */
}
```

Clarification: What I Do and Do Not Expect You to Know

Here 'know' means 'be able to do yourself'
'not know' means 'be able to read but not do yourself'

Things I Do expect you to know

- `assert( condition );`
  checks if `condition` is true; if `condition` is false the program is stopped with a helpful error message
- `malloc( size );`
  returns address of block of size bytes of memory (or `NULL` if not available); needs type cast to appropriate pointer type
- `free( pointer );`
  gives back block of memory indicated by `pointer` to the OS

Things I Don't expect you to know

- `printf( format, argument1, ... );`
  prints something, includes values of argument in this
- `scanf( format, &argument1, ...);`
  reads from keyboard, stores values in argument1 (and others)
  returns number of inputs or negative number in case of errors

Summary `malloc` and `free`

- `void *malloc(size_t size)`
  expects as argument number of bytes to be allocated
  returns address of continuous block of this size if possible
  returns `NULL` if not successful
- to allocate space for `n` items of type `type`
  `type *prt = (type *)malloc(sizeof(type)*n);`
- Always use `sizeof`,
  do not assume you know the size of a type.
- Always check if `malloc` returned NULL!
- Return space no longer needed to OS via `free(ptr);`
- Forgetting this leads to memory leaks.
- Prefer allocating a larger chunk of memory
  over many small ones to avoid memory fragmentation.

Multi-Dimensional Dynamic Arrays

What about multi-dimensional dynamic arrays?

Two options

1. one `malloc`, index calculations 'manually'
2. several `mallocs`, index calculations 'automatically'

Advantages

1. +easy `malloc` and `free`
   +very efficient
   +multi-dimensional array is one continuous block in memory
2. +looks exactly like static multi-dimensional array in use
   i. e., easy to use

Disadvantages

1. −index calculations more effort
   −does not look as natural
2. −`malloc` and `free` more work
   −a bit less efficient
   −memory fragmentation possible
   −location in memory unclear

Example: Two-Dimensional Array With One `malloc`

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */

int main(void) {
  int *a;
  int lines, columns, i, j;

  lines=3; /* stupid, boring, arbitrary */
  columns=7; /* this here, too */
  a = (int *)malloc(sizeof(int)*lines*columns); /* obtain space for 2-dim. array */
  assert( a != NULL ); /* and check we have it */
  for ( i=0; i<lines; i++ )
    for ( j=0; j<columns; j++ )
      a[i*columns+j] = 10*i + j; /* a[i][j]=10i+j */
  for ( i=0; i<lines*columns; i++ ) /* go through a continuously */
    printf("%02d ", a[i]); /* show contents of a[i] */
  free(a); /* return space to OS */
  return 0; /* exits program without error */
}
```

Output

00 01 02 03 04 05 06 10 11 12 13 14 15 16 20 21 22 23 24 25 26

Example: Three-Dimensional Array With One `malloc`

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */

int main(void) {
  int *a;
  int xsize, ysize, zsize, i, j, k;

  xsize=3; /* stupid, boring, arbitrary */
  ysize=5; /* this here, too */
  zsize=4; /* also this */
  a = (int *)malloc(sizeof(int)*xsize*ysize*zsize); /* obtain space for 3-dim. array */
  assert( a != NULL ); /* and check we have it */
  for ( i=0; i<xsize; i++ )
    for ( j=0; j<ysize; j++ )
      for ( k=0; k<zsize; k++ )
        a[i*ysize*zsize+j*zsize+k] = 100*i + 10*j + k; /* a[i][j][k]=100i+10j+k */
  for ( i=0; i<xsize*ysize*zsize; i++ ) /* go through a continuously */
    printf("%03d ", a[i]); /* show contents of a[i] */
  free(a); /* return space to OS */
  return 0; /* exits program without error */
}
```

Output 000 001 002 003 010 011 012 013 020 021 022 023 030 031 032 033 040 041 042 043 100 101 102 103 110 111 112 113 120 121 122 123 130 131 132 133 140 141 142 143 200 201 202 203 210 211 212 213 220 221 222 223 230 231 232 233 240 241 242 243

Example: Two-Dimensional Array With Multiple `mallocs`

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */
int main(void) {
  int **a;
  int lines, columns, i, j;

  lines=3; /* stupid, boring, arbitrary */
  columns=7; /* this here, too */
  a = (int **)malloc(sizeof(int *)*lines); /* obtain space for 1-dim. array of 1-dim. arrays */
  assert( a != NULL ); /* and check we have it */
  for ( i=0; i<lines; i++ ) { /* for each line */
    a[i] = (int *)malloc(sizeof(int)*columns); /* obtain space for 1-dim. array */
    assert( a[i] != NULL ); /* and check we have it */
  }
  for ( i=0; i<lines; i++ )
    for ( j=0; j<columns; j++ )
      a[i][j] = 10*i + j; /* a[i][j]=10i+j */
  for ( i=0; i<lines; i++ )
    for ( j=0; j<columns; j++ )
      printf("%02d ", a[i][j]); /* show contents of a[i][j] */
  for ( i=0; i<lines; i++ ) /* for each line */
    free( a[i] ); /* return space to OS */
  free(a); /* return space to OS */
  return 0; /* exits program without error */
}
```

Output 00 01 02 03 04 05 06 10 11 12 13 14 15 16 20 21 22 23 24 25 26

Example: Three-Dimensional Array With Multiple `mallocs`

```
#include <stdlib.h> /* defines malloc and free */
#include <assert.h> /* defines assert, supports safe programming */
#include <stdio.h> /* for input and output */
int main(void) {
  int ***a;
  int xsize, ysize, zsize, i, j, k;
  xsize=2; ysize=4; zsize=3; /* stupid, boring, arbitrary */
  a = (int ***)malloc(sizeof(int **)*xsize); /* obtain space for level 1 array */
  assert( a != NULL ); /* and check we have it */
  for ( i=0; i<xsize; i++ ) {
    a[i] = (int **)malloc(sizeof(int *)*ysize); /* obtain space for level 2 arrays */
    assert( a[i] != NULL ); /* and check we have it */
    for ( j=0; j<ysize; j++ ) {
      a[i][j] = (int *)malloc(sizeof(int)*zsize); /* obtain space for level 3 arrays */
      assert( a[i][j] != NULL ); /* and check we have it */
    }
  }
  for ( i=0; i<xsize; i++ )
    for ( j=0; j<ysize; j++ )
      for ( k=0; k<zsize; k++ ) {
        a[i][j][k] = 100*i + 10*j + k; /* a[i][j][k]=100i+10j+k */
        printf("%03d ", a[i][j][k]); /* show contents of a[i][j][k] */
      }
  for ( i=0; i<xsize; i++ ) {
    for ( j=0; j<ysize; j++ )
      free( a[i][j] ); /* return space to OS */
    free( a[i] ); /* return space to OS */
  }
  free(a); /* return space to OS */
  return 0; /* exits program without error */
}
```

Output 000 001 002 010 011 012 020 021 022 030 031 032 100 101 102 110 111 112 120 121 122 130 131 132

Looking Back on Some Algorithms We Discussed

- linear search
  Compare key with left-most entry.
  If equal return position else continue in remaining array.
- binary search
  Compare key with middle entry.
  If equal return position else
  if key smaller continue in left half else continue in right half.

Observation share algorithm pattern

1. Do something.
2. Continue with the same algorithm in a part of the original input.

Implementing Known Algorithms Differently

linear search

Compare key with
  left-most entry.
If equal return position
else continue in
  remaining array.

```
int linearSearch(int *keys, int key, int start, int end) {
   if ( start > end )
     return -1; /* empty array cannot contain key */
   if ( keys[start]==key )
     return start; /* key found */
   else
     return linearSearch(keys, key, start+1, end);
}
```

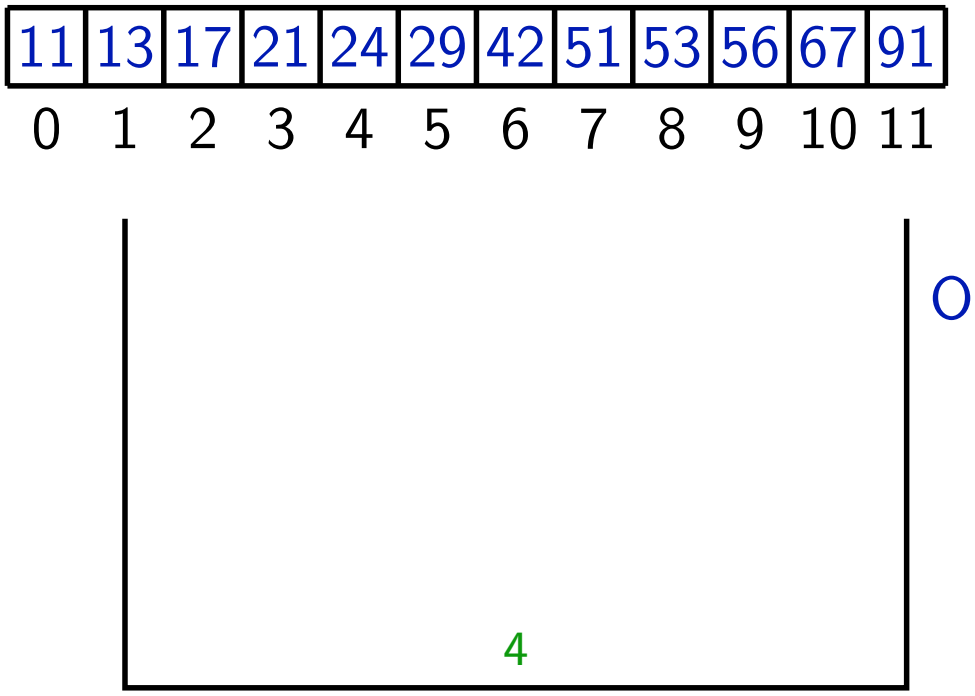
binary search

Compare key with middle entry.
If equal return position
else if key smaller
  continue in left half
  else continue in right half.

```
int binarySearch(int *keys, int key, int start, int end) {
   if ( start > end )
     return -1; /* empty array cannot contain key */
   if ( keys[start+((end-start)/2)]==key )
     return start+((end-start)/2); /* key found */
   else
     if ( keys[start+((end-start)/2)] < key )
       return binarySearch(keys, key,
         start+((end-start)/2)+1, end); /* search right */
     else
       return binarySearch(keys, key, start,
         start+((end-start)/2)-1); /* search left */
```

Example: Recursive binarySearch

```
int binarySearch(int *keys, int key, int start, int end) {
  if ( start > end )
    return -1; /* empty array cannot contain key */
  if ( keys[start+((end-start)/2)]==key )
    return start+((end-start)/2); /* key found */
  else
    if ( keys[start+((end-start)/2)] < key )
      return binarySearch(keys, key,
        start+((end-start)/2)+1, end); /* search right */
    else
      return binarySearch(keys, key, start,
        start+((end-start)/2)-1); /* search left */
```



Observation space on stack needed proportional to call depth
Requires care that call depth does not get too large

Consequence recursive `linearSearch` very bad idea

Summary Recursion

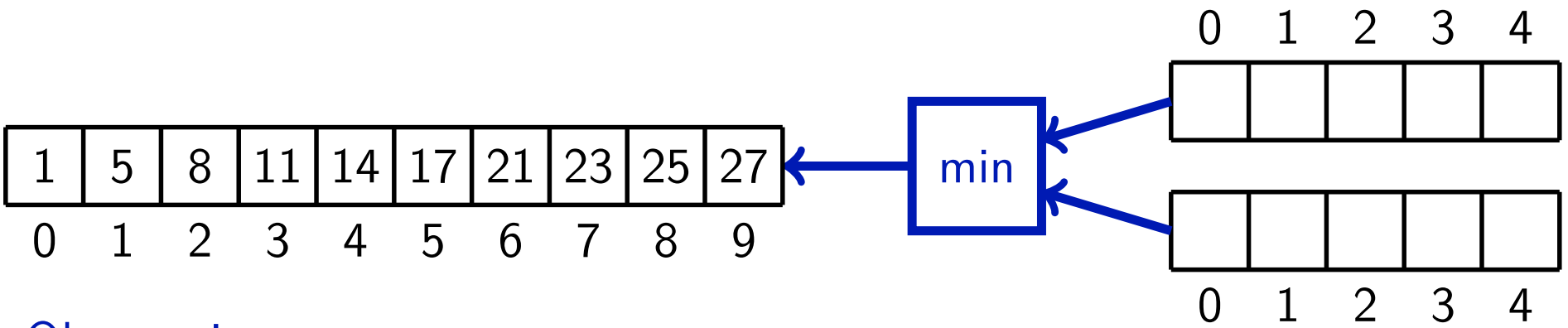
- recursion allows for simple, natural algorithms using pattern
  1. Check for trivial solution and stop if possible.
  2. Do something to make the problem smaller.
  3. Use the function itself to solve the smaller problem.
- recursion uses more spaces than is obvious
  due to space on call stack
- recursion requires care that call depth does not get too large
- when (obvious) iterative solution is available
  it is almost surely better
- recursive algorithms with only one recursive call at the very end can easily be replaced by iterative variant (and many compilers do that automatically)

Remember: Sorting

Can we sort as efficiently as HeapSort with a simpler algorithm using recursion?

An observation leading to an idea for a recursive sorting algorithm

two sorted sequences can be easily merged into one sorted sequence



Observations

- really simple and fast
- 'min' needs to be able to take care of special cases at the end
- extra space required, i. e., not in situ

Summary & Take Home Message

Things to remember

- dynamic multi-dimensional arrays
  - 'fragmented' and 'compact' version
  - index computations
- recursion
- stack and call depth
- recursive binary search
- merging sorted sequences

Take Home Message

- Dynamic arrays allow for responsible use of space.
- Index computations are not difficult.
- Recursive algorithms can be natural and simple.
- Recursion should only be used consciously.
- Recursion should not be used needlessly.

Lecture feedback `http://onlineted.com`